C# 프로그래밍

- 4 주차 (1강)

SEJONG UNIVERSITY

클래스 (class)
- 메소드
- 객체 복사하기
- this 키워드/생성자

출력 전용 매개변수: out

- 메소드에서 수행결과 값을 저장하기 위해 사용되는 매개변수
  - ✓ 메소드의 선언부와 호출부에 out 키워드 사용
  - ✓ 메소드는 하나 이상의 수행 결과 값을 호출자에게 전달 가능

```
static void Divide(int a, int b, out int quotient, out int remainder)
{
    quotient = a / b;
    remainder = a % b;
}

static void Main(string[] args)
{
    int a = 20;
    int b = 3;
    Divide(a, b, out int c, out int d);
    Console.WriteLine($"a:{a}, b:{b}, a/b:{c}, a%b:{d}");
}
```

out 키워드를
사용하여 코드 작성

출력 결과:
a:20, b:3, a/b:6, a%b:2

out 와 ref 키워드 사이의 차이점

- ref 키워드와 달리 out 키워드 사용 시 메소드에서 출력전용 매개변수에 값을 할당하지 않은 경우 컴파일 에러 발생
- out 키워드를 사용 시 메소드 호출 전에 미리 선언할 필요 없음

```
static void Divide(int a, int b, ref int quotient, ref int remainder)
{
    quotient = a / b;
    remainder = a % b;
}

static void Main(string[] args)
{
    int a = 20;
    int b = 3;
    int c=0, d=0;
    Divide(a, b, ref c, ref d);
    Console.WriteLine($"a:{a}, b:{b}, a/b:{c}, a%b:{d}");
}
```

ref 키워드를
사용하여 코드 작성

출력 결과:
a:20, b:3, a/b:6, a%b:2

메소드 오버로딩 (Overloading)

- 같은 이름을 사용하는 여러 개의 메소드를 하나의 클래스 안에 구현
  - ✓ 컴파일러는 매개변수의 수와 형식을 분석해 어떤 버전의 메소드 호출인지 확인
  - ✓ "Console.WriteLine()" 메소드는 총 18 버전을 오버로딩

```
public static void WriteLine(uint value);
public static void WriteLine(string format, params object?[]? arg);
public static void WriteLine();
public static void WriteLine(bool value);
public static void WriteLine(char[]? buffer);
public static void WriteLine(char[] buffer, int index, int count);
public static void WriteLine(decimal value);
public static void WriteLine(double value);
public static void WriteLine(ulong value);
public static void WriteLine(int value);
public static void WriteLine(object? value);
public static void WriteLine(float value);
public static void WriteLine(string? value);
public static void WriteLine(string format, object? arg0);
public static void WriteLine(string format, object? arg0, object? arg1);
public static void WriteLine(string format, object? arg0, object? arg1, object? arg2);
public static void WriteLine(long value);
public static void WriteLine(char value);
```

메소드 오버로딩 예제 코드

```
static int Plus(int a, int b)
{
    return a + b;
}

static int Plus(int a, int b, int c)
{
    return a + b + c;
}

static double Plus(double a, double b)
{
    return a + b;
}

static double Plus(int a, double b)
{
    return a + b;
}
```

```
static void Main(string[] args)
{
    Console.WriteLine(Plus(1,2));
    Console.WriteLine(Plus(1, 2, 3));
    Console.WriteLine(Plus(1.0, 2.4));
    Console.WriteLine(Plus(1, 2.4));
}
```

출력 결과:

```
3
6
3.4
3.4
```

가변 개수의 인수

- 데이터형이 같은 인수의 개수를 유연하게 사용하도록 메소드 정의
  - ✓ params 키워드와 배열을 이용해 선언
  - ✓ 매개변수의 개수가 유한한 경우는 오버로딩 사용

```
int sum = 0;          // 메소드 호출 시 인수의 수가 다양한 경우
sum = Sum(1, 2);
sum = Sum(1, 2, 3);
sum = Sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

```
static int Sum(int a, int b)        // 오버로딩을 통한 메소드 작성
{                                   // 매개변수 개수가 달라지면 지원 불가
    return a + b;
}

static int Sum(int a, int b, int c)
{
    return a + b + c;
}

static int Sum(int a, int b, int c, int d, int e, int f, int g, int h, int i, int j)
{
    return a + b + c + d + e + f + g + h + i + j;
}
```

```
// params 키워드 사용한 메소드
static int Sum(params int[] args)
{
    int sum = 0;
    for (int i = 0; i < args.Length; i++)
    {
        sum += args[i];
    }

    return sum;
}
```

가변 개수의 인수 관한 예제 코드

```
static int Sum(params int[] args)
{
    Console.Write("Summing... ");
    int sum = 0;
    for (int i = 0; i < args.Length; i++)
    {
        if (i > 0)
            Console.Write(", ");
        Console.Write(args[i]);

        sum += args[i];
    }
    Console.WriteLine();

    return sum;
}

static void Main(string[] args)
{
    Console.WriteLine($"Sum : {Sum(1, 2)}");
    Console.WriteLine($"Sum : {Sum(1, 2, 3)}");
    Console.WriteLine($"Sum : {Sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)}");
}
```

출력 결과:

```
Summing... 1, 2
Sum : 3
Summing... 1, 2, 3
Sum : 6
Summing... 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Sum : 55
```

명명된 인수

- 메소드를 호출할 때 인수의 이름에 근거해서 데이터를 할당
  - ✓ 메소드 선언은 수정 없이 그대로 사용
  - ✓ 메소드 호출할 때만 인수의 이름 뒤에 콜론(:)을 붙인 뒤 할당할 데이터를 넣어 줌
  - ✓ 인수가 많아 어느 매개변수인지 분간이 어려운 경우 명명된 인수 활용

```
static void PrintProfile(string name, string phone)
{
    Console.WriteLine("Name: {0}, Phone: {1}", name, phone);
}

static void Main(string[] args)
{
    // 명명된 인수 사용 시 인수의 순서는 상관없음
    PrintProfile(phone: "010-123-1234", name: "박찬호");
    PrintProfile(name: "박지성", phone: "010-111-1111");
    PrintProfile("박세리", "010-222-2222");
    PrintProfile("김연아", phone: "010-333-3333");
}
```

출력 결과:

```
Name: 박찬호, Phone: 010-123-1234
Name: 박지성, Phone: 010-111-1111
Name: 박세리, Phone: 010-222-2222
Name: 김연아, Phone: 010-333-3333
```

선택적 인수

- 메소드의 매개변수는 기본값을 가질 수 있음
  - ✓ 기본값을 가진 매개변수는 메소드를 호출할 때 해당 인수를 생략
- 선택적 인수 (Optional Argument)
  - ✓ 기본값을 가진 매개변수는 인수를 생략할지 선택할 수 있음
  - ✓ 선택적 인수는 항상 필수 인수 뒤에 위치

```
static void MyMethod_0(int a = 0)
{
    Console.WriteLine("{0}", a);
}

static void MyMethod_1(int a, int b = 0)
{
    Console.WriteLine("{0}, {1}", a, b);
}

static void MyMethod_2(int a, int b, int c = 10, int d = 20)
{
    Console.WriteLine("{0}, {1}, {2}, {3}", a, b, c, d);
}
```

```
static void Main(string[] args)
{
    MyMethod_1(3);
    MyMethod_1(3, 4);
}
```

선택적 인수 예제 코드

- 메소드의 매개변수가 많을 경우,
  - ✓ 메소드 호출 코드를 보고 어느 매개변수에 인수를 할당하는지 분간이 어려움
  - ✓ 명명된 인수를 활용
- 메소드 오버로딩과 함께 사용 시 혼란을 야기
  - ✓ 개발자가 분명하게 오버로딩을 사용할지 선택적 인수를 사용할지 결정 필요

```
static void PrintProfile(string name, string phone = "")
{
    Console.WriteLine($"Name:{name}, Phone:{phone}");
}

static void Main(string[] args)
{
    PrintProfile("중근");
    PrintProfile("관순", "010-555-5555");
    PrintProfile(name: "봉길");
    PrintProfile(name: "동주", phone: "010-777-7777");
}
```

출력 결과:

```
Name:중근, Phone:
Name:관순, Phone:010-555-5555
Name:봉길, Phone:
Name:동주, Phone:010-777-7777
```

로컬 함수 (Local Function)

- 메소드 안에서 선언되고 선언된 메소드 안에서만 사용되는 함수
  - ✓ 선언 방법은 메소드와 동일
  - ✓ 로컬 함수는 자신이 속한 메소드의 지역 변수를 사용할 수 있음

```
class SomeClass
{
    public void SomeMethod()        // 메소드 선언
    {
        int count = 0;
        SomeLocalFunction(1, 2);     // 로컬 함수 호출
        SomeLocalFunction(3, 4);

        void SomeLocalFunction(int a, int b)    // 로컬 함수 선언
        {
            Console.WriteLine($"count : {++count}");    // 지역 변수 count 사용
            Console.WriteLine($"a + b : {a + b}");
        }
    }
}
```

로컬 함수 예제 코드

```
static string ToLowerString(string input)
{
    var arr = input.ToCharArray();
    for(int i=0; i<arr.Length; i++)
    {
        arr[i] = ToLowerChar(i);
    }

    char ToLowerChar(int i) // 로컬 함수 선언
    {
        if (arr[i] < 65 || arr[i] > 90) // A ~ Z의 ASCII 값 : 65 ~ 90
            return arr[i];
        else    // a ~ z의 ASCII 값 : 97 ~ 122
            return (char)(arr[i] + 32); // 지역변수 arr 사용
    }

    return new string(arr);
}
```

```
static void Main(string[] args)
{
    Console.WriteLine(ToLowerString("Hello!"));
    Console.WriteLine(ToLowerString("Good Morning."));
    Console.WriteLine(ToLowerString("Goodbye."));
}
```

출력 결과:

```
hello!
good morning.
goodbye.
```

클래스 (class)

- 메소드
- 객체 복사하기
- this 키워드/생성자

객체 복사: 얕은 복사

- 클래스는 참조 형식이고, 객체 사이의 얕은 복사는 스택에 있는 참조를 복사

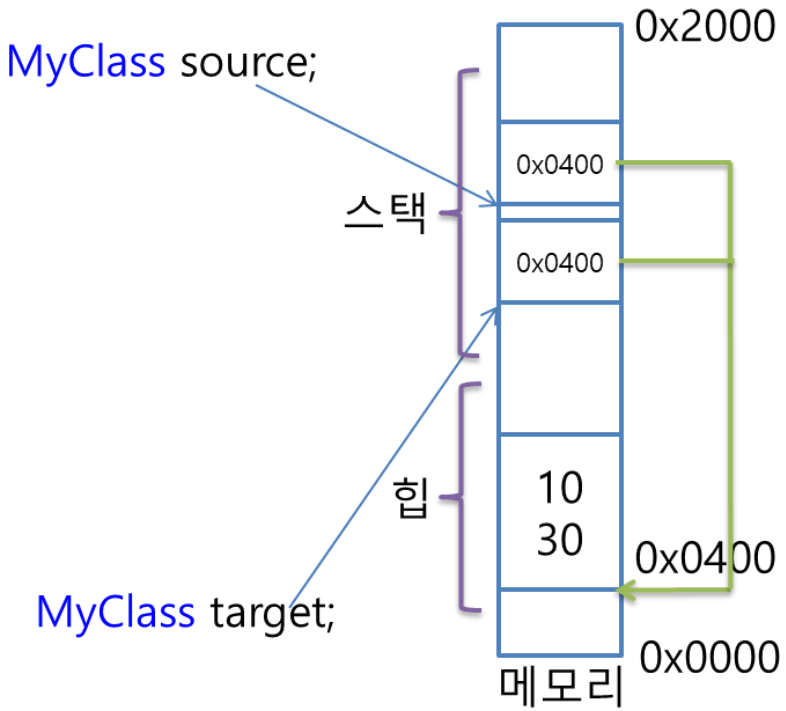
```
class MyClass
{
    public int MyField1;
    public int MyField2;
}

class MainApp
{
    static void Main(string[] args)
    {
        MyClass source = new MyClass();
        source.MyField1 = 10;
        source.MyField2 = 20;

        MyClass target = source; // 실제 객체가 아닌 스택에 있는 참조를
        target.MyField2 = 30;

        Console.WriteLine("{0} {1}", source.MyField1, source.MyField2);
        Console.WriteLine("{0} {1}", target.MyField1, target.MyField2);
    }
}
```

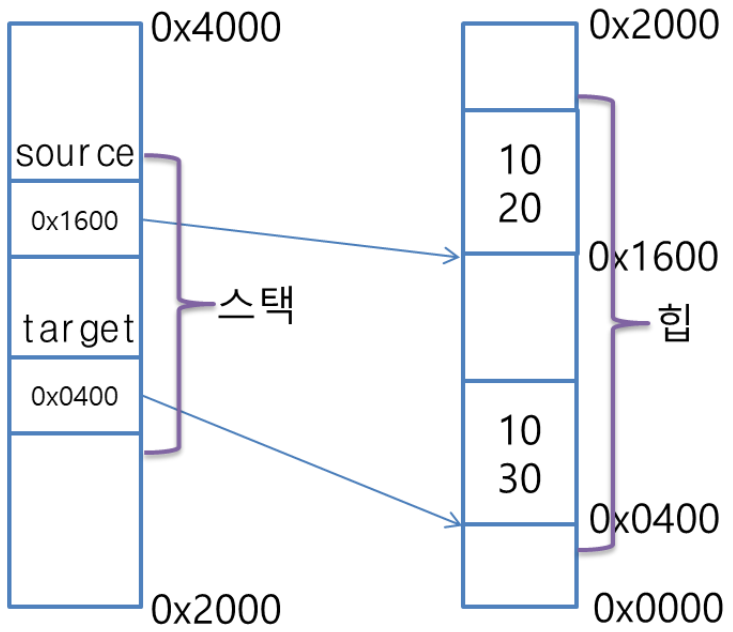
출력 결과:
10 30
10 30



객체 복사: 깊은 복사

- 깊은 복사는 두 객체 각각 힙에 서로 다른 공간이 있지만 같은 값을 포함
  - ✓ 아래 그림처럼, target 이 힙에 보관되어 있는 내용을 source 로부터 복사
  - ✓ 별도의 힙 공간에 객체를 보관
  - ✓ C#에서는 깊은 복사를 자동으로 해주는 구문이 없음

```
class MyClass
{
    public int MyField1;
    public int MyField2;
    public MyClass DeepCopy()
    { // 객체를 힙에 새로 할당 후, 자신의 멤버를 일일이 복사
        MyClass newCopy = new MyClass();
        newCopy.MyField1 = this.MyField1;
        newCopy.MyField2 = this.MyField2;
        return newCopy;
    }
}
```



얕은/깊은 복사 예제 코드

```
class MyClass
{
    public int MyField1;
    public int MyField2;
    public MyClass DeepCopy()
    { // 객체를 힙에 새로 할당 후,
      // 자신의 멤버를 일일이 복사
        MyClass newCopy = new MyClass();
        newCopy.MyField1 = this.MyField1;
        newCopy.MyField2 = this.MyField2;
        return newCopy;
    }
}
```

출력 결과:

```
Shallow Copy
10 30
10 30
Deep Copy
10 20
10 30
```

```
static void Main(string[] args)
{
    Console.WriteLine("Shallow Copy");
    {
        MyClass source = new MyClass();
        source.MyField1 = 10;
        source.MyField2 = 20;
        MyClass target = source; // 얕은 복사
        target.MyField2 = 30;

        Console.WriteLine("{0} {1}", source.MyField1, source.MyField2);
        Console.WriteLine("{0} {1}", target.MyField1, target.MyField2);
    }

    Console.WriteLine("Deep Copy");
    {
        MyClass source = new MyClass();
        source.MyField1 = 10;
        source.MyField2 = 20;
        MyClass target = source.DeepCopy(); // 깊은 복사
        target.MyField2 = 30;

        Console.WriteLine("{0} {1}", source.MyField1, source.MyField2);
        Console.WriteLine("{0} {1}", target.MyField1, target.MyField2);
    }
}
```

this 키워드

```
class Employee
{
    private string Name;
    private string Position;

    public void SetName(string Name)
    {
        this.Name = Name;
    }

    public string GetName()
    {
        return Name;
    }

    public void SetPosition(string Position)
    {
        this.Position = Position;
    }

    public string GetPosition()
    {
        return this.Position;
    }
}
```

- 객체 내부에서 자신의 멤버 접근할 때 this 키워드 사용

```
static void Main(string[] args)
{
    Employee pooh = new Employee();
    pooh.SetName("Pooh");
    pooh.SetPosition("Waiter");
    Console.WriteLine($"{pooh.GetName()} {pooh.GetPosition()}");

    Employee tigger = new Employee();
    tiger.SetName("Tiger");
    tiger.SetPosition("Cleaner");
    Console.WriteLine($"{tiger.GetName()} {tiger.GetPosition()}");
}
```

출력 결과:
Pooh Waiter
Tiger Cleaner

this() 생성자

- this() 생성자를 이용해 생성자 메소드에 중복된 코드를 줄일 수 있음
  - ✓ this() 는 자기 자신의 생성자를 가리킴
  - ✓ 생성자의 코드 블록 내부가 아닌 앞쪽에서만 사용 가능

오버로딩 된 생성자

```
class MyClass
{
    int a, b, c;
    public MyClass()
    {
        this.a = 5425;
    }
    public MyClass(int b)
    {
        this.a = 5425;
        this.b = b;
    }
    public MyClass(int b, int c)
    {
        this.a = 5425;
        this.b = b;
        this.c = c;
    }
}
```

MyClass() 와 중복 코드

MyClass(int b) 와 중복 코드

this () 생성자를 이용한 오버로딩 된 생성자

```
class MyClass
{
    int a, b, c;

    public MyClass()
    {
        this.a = 5425;
    }

    public MyClass(int b) : this()
    {
        this.b = b;
    }

    public MyClass(int b, int c) : this(b)
    {
        this.c = c;
    }
}
```

this( )는 MyClass( )를 호출

this(int)는 MyClass(int)를 호출

this() 생성자 예제 코드

```
class MyClass
{
    int a, b, c;
    public MyClass()
    {
        this.a = 5425;
        Console.WriteLine("MyClass()");
    }

    public MyClass(int b) : this()
    {
        this.b = b;
        Console.WriteLine($"MyClass({b})");
    }

    public MyClass(int b, int c) : this(b)
    {
        this.c = c;
        Console.WriteLine($"MyClass({b}, {c})");
    }

    public void PrintFields()
    {
        Console.WriteLine($"a:{a}, b:{b}, c:{c}");
    }
}
```

```
class MainApp
{
    static void Main(string[] args)
    {
        MyClass a = new MyClass();
        a.PrintFields();
        Console.WriteLine();

        MyClass b = new MyClass(1);
        b.PrintFields();
        Console.WriteLine();

        MyClass c = new MyClass(10, 20);
        c.PrintFields();
    }
}
```

출력 결과:

```
MyClass()
a:5425, b:0, c:0

MyClass()
MyClass(1)
a:5425, b:1, c:0

MyClass()
MyClass(10)
MyClass(10, 20)
a:5425, b:10, c:20
```

객체지향 프로그래밍

- 은닉성 (캡슐화)
- 상속성
- 다형성

SEJONG UNIVERSITY

은닉성 (캡슐화) 의미

- 클래스의 사용자에게 필요한 최소의 기능만 노출하고 내부를 감추는 것
- 예를 들어 선풍기를 생각해 보면,
  - ✓ 버튼 3개(바람세기 조절)와 다이얼 2개(회전과 타이머)를 사용자에게 제공
  - ✓ 선풍기 케이스 안에 회로와 배선 등은 사용자에게 감춤
  - ✓ 만약, 선풍기의 회로와 배선을 사용자가 조작하도록 노출한다면 문제 발생
- 캡슐화가 잘 된 클래스
  - ✓ 클래스의 이름 자체에서 제공되는 기능을 대략 파악 가능
  - ✓ 외부로 제공해야 할 기능에 대해서만 노출

접근 제한자 (Access Modifier)

- 감추고 싶은 것은 감추고, 보여주고 싶은 것은 보여주도록 코드를 수식
  - ✓ 클래스 안에 필드, 메소드, 프로퍼티 등 모든 요소에 사용 가능
  - ✓ C#에서는 총 여섯 가지 접근 제한자 제공

접근 제한자	설명
public	클래스의 내부/외부 모든 곳에서 접근 가능
protected	클래스의 외부에서 접근 불가능, but 파생 클래스에서 접근 가능
private	클래스의 내부에서만 접근 가능
internal	같은 어셈블리에 있는 코드에서만 public 이고, 다른 어셈블리에 대해 private 수준의 접근성
protected internal	같은 어셈블리에 있는 코드에서만 protected 이고, 다른 어셈블리에 대해 private 수준의 접근성
private protected	같은 어셈블리에 있는 클래스에서 상속받은 클래스 내부에서만 접근 가능

접근 제한자 사용 형식

- 어셈블리 (Assembly)
  - ✓ .NET 에서는 EXE 또는 DLL 형식의 C# 파일을 어셈블리라고 함
  - ✓ 이론 상 어셈블리는 하나 이상의 모듈로 구성 (모듈 하나당 한 개의 파일)
  - ✓ 일반적으로 1개의 (EXE/DLL 모듈) 파일로 구성된 어셈블리가 사용됨
- 접근 제한자로 수식하지 않은 클래스의 멤버는 무조건 private 으로 자동 지정

```
class MainApp
{
    private     int MyField_1;
    protected   int MyField_2;
                int MyField_3;  // 접근 제한자로 수식하지 않으면 private 과 같은 공개수준

    public int MyMethod_1()
    {
        // ...
    }
    internal void MyMethod_2 ()
    {
        // ...
    }
}
```

접근 제한자 예제 코드

출력 결과:
```
Turn on water : 20
Turn on water : -2
Out of temperature range
```

```
class WaterHeater
{
    protected int temperature;
    public void SetTemperature(int temperature)
    { // -5 ~ 42 사이의 값만 할당하고, 그 외의 값은 예외발생
        if (temperature < -5 || temperature > 42)
        {
            throw new Exception("Out of temperature range");
        }

        // temperature 필드는 클래스 내부에서 접근 가능
        this.temperature = temperature;
    }

    internal void TurnOnWater()
    {
        Console.WriteLine($"Turn on water : {temperature}");
    }
}
```

```
static void Main(string[] args)
{
    try
    {
        WaterHeater heater = new WaterHeater();
        heater.SetTemperature(20);
        heater.TurnOnWater();

        heater.SetTemperature(-2);
        heater.TurnOnWater();

        // 42 값보다 큰 값이 인수로 사용, 예외발생
        heater.SetTemperature(50);
        heater.TurnOnWater();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Thank you!